3.4 Modular Structural Operational Semantics (MSOS)

Modular structural operational semantics (MSOS) was introduced, as its name implies, to address the non-modularity aspects of (big-step and/or small-step) SOS. There are several reasons why big-step and small-step SOS are non-modular, as well as several facets of non-modularity in general. These are discussed in detail in Section 3.9. In short, a definitional framework is *non-modular* when, in order to add a new feature to an existing language or calculus, one needs to revisit and change some or all of the already defined unrelated features. For example, suppose that one wants to add an output statement to IMP, as we do in IMP++ in Section 3.8. Then one needs to add a new semantic component in the IMP configurations, both in the big-step and in the small-step SOS definitions, to hold the output "buffer". That means, in particular, that *all* the existing big-step and/or small-step SOS rules of IMP need to change, which is, of course, very inconvenient.

Before we get into the technicalities of MSOS, one natural question to address is why we need modularity of language definitions. One may argue that defining a programming language is a major endeavor, done once and for all, so having to go through the defining rules many times is, after all, not such a bad idea, because it gives one the chance to find and fix potential errors in them. Here are several reasons why modularity is desirable in language definitions:

- Having to modify many or all rules whenever a new rule is added that modifies the structure of the configuration is actually more error prone than it may seem, because rules become heavier to read and debug; for example, one can write σ instead of σ' in a right-hand side of a rule and a different or wrong language is defined.

- When designing a new language, as opposed to an existing well-understood language, one needs to experiment with features and combinations of features; having to do lots of unrelated changes whenever a new feature is added to or removed from the language burdens the language designer with boring tasks taking considerable time that could have been otherwise spent on actual interesting language design issues.

- There is a plethora of domain-specific languages these days, generated by the need to abstract away from low-level programming language details to important, domain-specific aspects of the application. Therefore, there is a need for rapid language design and experimentation for various domains. Moreover, domain-specific languages tend to be very dynamic, being added or removed features frequently as the domain knowledge evolves. It would be very nice to have the possibility to "drag-and-drop" language features in one's language, such as functions, exceptions, objects, etc.; however, in order for that to be possible, modularity is crucial.

Whether MSOS gives us such a desired and general framework for modular language design is and will probably always be open for debate. However, at our knowledge MSOS is the first framework that explicitly recognizes the importance of modular language design and provides explicit support to achieve it in the context of SOS. Reduction semantics with evaluation contexts (see Section 3.8.1) was actually proposed before MSOS and also offers modularity in language semantic definitions, but its modularity comes as a consequence of a different way to propagate reductions through language constructs and not as an explicit goal that it strives to achieve.

There are both big-step and small-step variants of MSOS, but we discuss only small-step MSOS. We actually generically call MSOS the small-step, implicitly-modular variant of MSOS (see Section 3.4.3). To bring modularity to SOS, MSOS proposes the following:

- Separate the syntax (i.e., the fragment of program under consideration) from the non-syntactic components in configurations, and treat them differently, as explained below;
- Make the transitions relate syntax to syntax only (as opposed to configurations), and "hide" the non-syntactic components in a transition label, as explained below;
- Encode in the transition label all the changes in the non-syntactic components of the configuration that need to be applied together with the syntactic reduction given by the transition;
- Use specialized notation in transition labels together with a discipline to refer to the various semantic components and to say that some of them stay unchanged; also, labels can be explicitly or implicitly shared by the conditions and the conclusion of a rule, elegantly capturing the idea that "changes are propagated" through desired language constructs.

A transition in MSOS is of the form

$$P \xrightarrow{\Delta} P'$$

where P and P' are programs or fragments of programs and Δ is a *label describing the semantic configuration components both before and after the transition*. Specifically, Δ is a record containing fields denoting the semantic components of the configuration. The preferred notation in MSOS for stating that in label Δ the semantic component associated to the field name field *before* the transition takes place is α is $\Delta = \{\mathsf{field} = \alpha, \ldots\}$. Similarly, the preferred notation for stating that the semantic component associated to field field *after* the transition takes place is β is $\Delta = \{\mathsf{field'} = \beta, \ldots\}$ (the field name is primed). For example, the second MSOS rule for variable assignment (when the assigned arithmetic expression is already evaluated) is (this is rule (MSOS-Asgn) in Figure 3.21):

$$x := i \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[i/x],\ \ldots\}} \texttt{skip}$$

It is easy to "desugar" the rule above into a more familiar SOS rule of the form:

$$\langle x := i, \sigma \rangle \to \langle\, \texttt{skip}\,, \sigma[i/x] \rangle$$

The above is precisely the rule (SmallStep-Asgn) in the small-step SOS of IMP (see Figure 3.15). The MSOS rule is actually more modular than the SOS one, because of the "...", which say that "everything else stays unchanged" in the configuration. For example, if we want to extend the language with an output statement as we do within IMP++ in Section 3.8, then a new component, the "output buffer", needs to be added to the configuration to collect and store the output. Then the SOS rule above needs to be changed into a rule of the form

$$\langle x := i, \sigma, \omega \rangle \to \langle\, \texttt{skip}\,, \sigma[i/x], \omega \rangle$$

where ω is the output buffer, which stays unchanged during the variable assignment operation, while the MSOS rule does not need to be touched.

To impose a better discipline on the use of labels, at the same time making the notation even more compact, MSOS splits the fields into three categories: *read-only*, *read-write*, and *write-only*. The field state above was read-write, meaning that the transition label can both read its value before the transition takes place and write its value after the transition takes place. Unlike the state, which needs to be both read and written, there are semantic configuration components that only need to be written, as well as ones that only need to be read. In these cases, it is recommended to use

write-only or read-only fields. Read-only fields are only inspected by the rule, but not modified, so they only appear unprimed in labels. For example, when reading the location of a variable in an environment, the environment is not modified. We do not discuss read-only fields further here. The write-only fields are used to record data that is not analyzable during the execution of the program, such as the output or the trace. Their names are always primed and they have a free monoid semantics — everything written on them is actually added to the end. Consider, for example, the extension of IMP with an output (or print) statement in Section 3.8, whose MSOS second rule (after the argument is evaluated, namely rule (MSOS-PRINT) in Section 3.8) is:

$$\texttt{print}\ i \xrightarrow{\{\mathsf{output'}=i,\ \ldots\}} \texttt{skip}$$

Compare the rule above with the one below, which uses a read-write attribute instead:

$$\texttt{print}\ i \xrightarrow{\{\mathsf{output}=\omega,\ \mathsf{output'}=\omega{:}i,\ \ldots\}} \texttt{skip}$$

Indeed, mentioning the ω is unnecessary, error-prone (e.g., one may forget to add it to the primed field, or may write $i : \omega$ instead of $\omega : i$), and non-modular (e.g., one may want to change the binary construct of the output monoid, say to write $\omega \cdot i$ instead of $\omega : i$, etc.).

MSOS achieves modularity in two ways: 1) by making intensive use of the record comprehension notation "...", which, as discussed, indicates that more fields could follow but that they are not of interest for this transition, in particular that they stay unchanged after the application of the transition; and 2) by reusing the same label or portion of label both in the condition and the conclusion of a rule (in particular, if record comprehension is used in both the condition and the conclusion of an MSOS rule, then all the occurrences of "..." stand for the same portion of label, that is, the same fields bound to the same semantic components. For example, the following MSOS rules for first-statement reduction in sequential composition are equivalent and say that all the configuration changes generated by reducing s_1 to s_1' are propagated when reducing $s_1 \ ; \ s_2$ to $s_1' \ ; \ s_2$:

$$\frac{s_1 \xrightarrow{\Delta} s_1'}{s_1 \ ; \ s_2 \xrightarrow{\Delta} s_1' \ ; \ s_2}$$

$$\frac{s_1 \xrightarrow{\{\ldots\}} s_1'}{s_1 \ ; \ s_2 \xrightarrow{\{\ldots\}} s_1' \ ; \ s_2}$$

$$\frac{s_1 \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} s_1'}{s_1 \ ; \ s_2 \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} s_1' \ ; \ s_2}$$

Indeed, advancing one step the first statement in a sequential composition of statements has the same effects on the configuration as if the statement was advanced the same one step in isolation, without the other statement involved; said differently, the side effects are all properly propagated.

MSOS (the implicitly-modular variant of it, see Section 3.4.3) has been refined to actually allow for dropping such redundant labels like above from rules. In other words, if a label is missing from a transition then the "implicit label" is assumed: if the rule is unconditional then the implicit label

is the identity label (in which the primed fields have the values as the corresponding unprimed ones, etc.), while if the rule is conditional, then the condition and the conclusion transitions share the same label, that is, they perform the same changes on the semantic components of the configuration. With this new notational convention, the most elegant way to write the rule above in MSOS is:

$$\frac{s_1 \to s_1'}{s_1 \; ; \; s_2 \to s_1' \; ; \; s_2}$$

This is precisely the rule (MSOS-Seq-Arg1) in Figure 3.21, part of the MSOS semantics of IMP.

One of the important merits of MSOS is that it captures formally many of the "tricks" that language designers informally use to avoid writing awkward and heavy SOS definitions.

Additional notational shortcuts are welcome in MSOS if properly explained and made locally rigorous, without having to rely on other rules. For example, the author of MSOS finds the rule

$$x \xrightarrow{\{\mathsf{state},\ \ldots\}} \mathsf{state}(x)$$

to be an acceptable variant of the lookup rule:

$$x \xrightarrow{\{\mathsf{state}=\sigma,\ \ldots\}} \sigma(x)$$

despite the fact that, strictly speaking, $\mathsf{state}(x)$ does not make sense by itself (recall that state is a field name, not the state function) and that field names are expected to be paired with their semantic component in labels. Nevertheless, there is only one way to make sense of this rule, namely to replace any use of state by its semantic contents, which therefore does not need to be mentioned.

A major goal when using MSOS to define languages or calculi is to write on the labels as little information as possible and to use the implicit conventions for the missing information. That is because everything written on labels may work against modularity if the language is later on extended or simplified. As an extreme case, if one uses only read/write fields in labels and mentions all the fields together with all their semantic contents on every single label, then MSOS becomes conventional SOS and therefore suffers from the same limitations as SOS in what regards modularity.

Recall the rules in Figure 3.16 for deriving the transitive closure $\to^\star$ of the small-step SOS relation $\to$. In order for two consecutive transitions to "compose", the source configuration of the second had to be identical to the target configuration of the first. A similar property must also hold in MSOS, otherwise one may derive inconsistent computations. This process is explained in MSOS by making use of category theory (Section 2.10), associating MSOS labels with morphisms in a special category and then using the morphism composition mechanism of category theory.

However, category theory is not needed in order to understand how MSOS works in practice. A simple way to explain its label composition is by translating, or "desugaring" MSOS definitions into SOS, as we implicitly implied when we discussed the MSOS rule for variable assignment above. Indeed, once one knows all the fields in the labels, which happens once a language definition is complete, one can automatically associate a standard small-step SOS definition to the MSOS one by replacing each MSOS rule with an SOS rule over configurations containing, besides the syntactic contents, the complete semantic contents extracted from the notational conventions in the label. The resulting SOS configurations will not have fields anymore, but will nevertheless contain all the semantic information encoded by them. For example, in the context of a language containing only a state and an output buffer as semantic components in its configuration, the four rules discussed

above for variable assignment, output, sequential composition, and lookup desugar into the following conventional SOS rules, as expected (listed in the discussed order):

$$\langle x := i, \sigma, \omega \rangle \to \langle \texttt{skip}, \sigma[i/x], \omega \rangle$$

$$\langle \texttt{print}\ i, \sigma, \omega \rangle \to \langle \texttt{skip}, \sigma, \omega : i \rangle$$

$$\frac{\langle s_1, \sigma, \omega \rangle \to \langle s_1', \sigma', \omega' \rangle}{\langle s_1 \ ; \ s_2, \sigma, \omega \rangle \to \langle s_1' \ ; \ s_2, \sigma', \omega' \rangle}$$

$$\langle x, \sigma, \omega \rangle \to \langle \sigma(x), \sigma, \omega \rangle$$

Note that for unconditional MSOS rules the meaning of the missing label fields is "stay unchanged", while in the case of conditional rules the meaning of the missing fields is "same changes in conclusion as in the condition". In order for all the changes explicitly or implicitly specified by MSOS rules to apply, one also needs to provide an initial state for all the attributes, or in terms of SOS, an initial configuration. The initial configuration is often left unspecified in MSOS or SOS "paper" language definitions, but it needs to be explicitly given when one is concerned with executing the semantics. In our SOS definition of IMP in Section 3.3.2 (see Figure 3.15), we created the appropriate initial configuration in which the top-level statement was executed using the proof system itself, more precisely the rule (SmallStep-Vars) created a configuration holding a statement and a state from a configuration holding only the program. That is not possible in MSOS, because MSOS assumes that the structure of the label record does not change dynamically as the rules are applied. Instead, it assumes all the attributes given and fixed. Therefore, one has to explicitly state the initial values corresponding to each attribute in the initial state. However, in practice those initial values are understood and, consequently, we do not bother defining them. For example, if an attribute holds a list or a set, then its initial value is the empty list or set; if it holds a partial function, then its initial value is the partial function undefined everywhere; etc.

This way of regarding MSOS as a convenient "front-end" to SOS also supports the introduction of further notational conventions in MSOS if desired, like the one using $\mathsf{state}(x)$ instead of $\sigma(x)$ in the right-hand side of the transition above, as far as one clearly explains how such conventions are desugared when going from MSOS to SOS. Finally, the translation of MSOS into SOS also allows MSOS to "borrow" from SOS the reflexive/transitive closure $\to^\star$ of the one-step relation.

3.4.1 The MSOS of IMP

Figures 3.20 and 3.21 show the MSOS definition of IMP. There is not much to comment on the MSOS rules in these figures, except, perhaps, to note how compact and elegant they are compared to the corresponding SOS definition in Figures 3.14 and 3.15. Except for the three rules (MSOS-Lookup), (MSOS-Asgn), and (MSOS-Vars), which make use of labels, they are as compact as they can be in any SOS-like setting for any language including the defined constructs. Also, the above-mentioned three rules only mention those components from the labels that they really need, so they allow for possible extensions of the language, as we will do with IMP++ in Section 3.8.

The rule (MSOS-Vars) is somehow different from the other rules that need the information in the label, in that it uses a read-write attribute but it only writes it without reading it. This is indeed possible in MSOS, though it tells us that the type of an attribute cannot be necessarily

$$x \xrightarrow{\{\mathsf{state}=\sigma,\ldots\}} \sigma(x) \qquad \text{(MSOS-Lookup)}$$

$$\frac{a_1 \to a_1'}{a_1 + a_2 \to a_1' + a_2} \qquad \text{(MSOS-Add-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 + a_2 \to a_1 + a_2'} \qquad \text{(MSOS-Add-Arg2)}$$

$$i_1 + i_2 \to i_1 +_{Int} i_2 \qquad \text{(MSOS-Add)}$$

$$\frac{a_1 \to a_1'}{a_1 \,/\, a_2 \to a_1' \,/\, a_2} \qquad \text{(MSOS-Div-Arg1)}$$

$$\frac{a_2 \to a_2'}{a_1 \,/\, a_2 \to a_1 \,/\, a_2'} \qquad \text{(MSOS-Div-Arg2)}$$

$$i_1 \,/\, i_2 \to i_1 \,/_{Int}\, i_2 \quad \text{when } i_2 \neq 0 \qquad \text{(MSOS-Div)}$$

$$\frac{a_1 \to a_1'}{a_1 \texttt{<=} a_2 \to a_1' \texttt{<=} a_2} \qquad \text{(MSOS-Leq-Arg1)}$$

$$\frac{a_2 \to a_2'}{i_1 \texttt{<=} a_2 \to i_1 \texttt{<=} a_2'} \qquad \text{(MSOS-Leq-Arg2)}$$

$$i_1 \texttt{<=} i_2 \to i_1 \leq_{Int} i_2 \qquad \text{(MSOS-Leq)}$$

$$\frac{b \to b'}{\texttt{not}\, b \to \texttt{not}\, b'} \qquad \text{(MSOS-Not-Arg)}$$

$$\texttt{not true} \to \texttt{false} \qquad \text{(MSOS-Not-True)}$$

$$\texttt{not false} \to \texttt{true} \qquad \text{(MSOS-Not-False)}$$

$$\frac{b_1 \to b_1'}{b_1 \,\texttt{and}\, b_2 \to b_1' \,\texttt{and}\, b_2} \qquad \text{(MSOS-And-Arg1)}$$

$$\texttt{false and}\, b_2 \to \texttt{false} \qquad \text{(MSOS-And-False)}$$

$$\texttt{true and}\, b_2 \to b_2 \qquad \text{(MSOS-And-True)}$$

Figure 3.20: MSOS(IMP) — MSOS of IMP Expressions ($i_1, i_2 \in \mathit{Int}$; $x \in \mathit{Id}$; $a_1, a_1', a_2, a_2' \in \mathit{AExp}$; $b, b', b_1, b_1', b_2 \in \mathit{BExp}$; $\sigma \in \mathit{State}$).

$$\frac{a \to a'}{x := a \to x := a'} \qquad \text{(MSOS-Asgn-Arg2)}$$

$$x := i \xrightarrow{\{\text{state}=\sigma,\ \text{state}'=\sigma[i/x],\ \ldots\}} \texttt{skip} \qquad \text{(MSOS-Asgn)}$$

$$\frac{s_1 \to s_1'}{s_1 \,;\, s_2 \to s_1' \,;\, s_2} \qquad \text{(MSOS-Seq-Arg1)}$$

$$\texttt{skip} \,;\, s_2 \to s_2 \qquad \text{(MSOS-Seq-Skip)}$$

$$\frac{b \to b'}{\texttt{if}\, b \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \to \texttt{if}\, b' \,\texttt{then}\, s_1 \,\texttt{else}\, s_2} \qquad \text{(MSOS-If-Arg1)}$$

$$\texttt{if true then}\, s_1 \,\texttt{else}\, s_2 \to s_1 \qquad \text{(MSOS-If-True)}$$

$$\texttt{if false then}\, s_1 \,\texttt{else}\, s_2 \to s_2 \qquad \text{(MSOS-If-False)}$$

$$\texttt{while}\, b \,\texttt{do}\, s \to \texttt{if}\, b \,\texttt{then}\, (s \,;\, \texttt{while}\, b \,\texttt{do}\, s) \,\texttt{else skip} \qquad \text{(MSOS-While)}$$

$$\texttt{vars}\, xl \,;\, s \xrightarrow{\{\text{state}'= xl \mapsto 0, \ldots\}} s \qquad \text{(MSOS-Vars)}$$

Figure 3.21: MSOS(IMP) — MSOS of IMP Statements ($i \in \mathit{Int}$; $x \in \mathit{Id}$; $xl \in \mathbf{List}\{\mathit{Id}\}$;$a, a' \in \mathit{AExp}$; $b, b' \in \mathit{BExp}$; $s, s_1, s_1', s_2 \in \mathit{Stmt}$; $\sigma \in \mathit{State}$).

inferred from the way it is used in rules. One should explicitly clarify the type of each attribute before one gives the actual MSOS rules. Indeed, if the type of the output attribute in the MSOS rules for output above (and also in Section 3.8) were read-write, then the rules would wrongly imply that the output buffer will only store the last output value, the previous ones being lost (this could be a desirable semantics in some cases, but not in our case).

Since the MSOS proof system in Figures 3.20 and 3.21 translates, following the informal procedure described above, in the SOS proof system in Figures 3.14 and 3.15, basically all the small-step SOS intuitions and discussions for IMP in Section 3.3.2 carry over here almost unchanged. In particular:

Definition 18. *We say that $C \to C'$ is derivable with the MSOS proof system in Figures 3.20 and 3.21, written* $\text{MSOS(IMP)} \vdash C \to C'$*, iff* $\text{SmallStep(IMP)} \vdash C \to C'$ *(using the proof system in Figures 3.14 and 3.15). Similarly,* $\text{MSOS(IMP)} \vdash C \to^\star C'$ *iff* $\text{SmallStep(IMP)} \vdash C \to^\star C'$.

Exercise 51. *Change the MSOS rules for* / *in Figure 3.20 so that it short-circuits when its numerator evaluates to* 0.

Exercise 52. *Change the MSOS rules of the* IMP *conjunction in Figure 3.20 so that it is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments.*

Exercise 53. *Give an alternative MSOS definition of* `while` *which wastes no computational step.*

Exercise 54. *Add* `error` *expressions and statements and modify the MSOS of* IMP *in Figures 3.20 and 3.21 to allow derivations of sequents whose right-hand side configurations contain* `error` *as their syntactic component when a division by zero takes place. For the time being, just follow the same non-modular approach as in the case of SOS (Exercise 44); we are going to discuss a more modular approach to handle errors in MSOS in Section 3.8.*

Note, however, that MSOS is more syntactic in nature than SOS, in that each of its reduction rules requires syntactic terms in both sides of the transition relation. In particular, that means that, unlike in SOS (see Exercise 42), in MSOS one does not have the option to "dissolve" statements from configurations anymore. Instead, one needs to reduce them to `skip` or some similar syntactic constant; if the original language did not have such a constant then one needs to invent one and add it to the original language or calculus syntax.

3.4.2 MSOS in Rewriting Logic

Like big-step and small-step SOS, we can also associate a conditional rewrite rule to each MSOS rule and hereby obtain a rewriting logic theory that faithfully (i.e., step-for-step) captures the MSOS definition. There could be different ways to do this. One way to do it is to first desugar the MSOS definition into a step-for-step equivalent small-step SOS definition as discussed above, and then use the faithful embedding of small-step SOS into rewriting logic discussed in Section 3.3.3. The problem with this approach is that the resulting small-step SOS definition, and implicitly the resulting rewriting logic definition, lack the modularity of the original MSOS definition. In other words, if one wanted to extend the MSOS definition with rules that would require global changes to its corresponding SOS definition (e.g., ones adding new semantic components into the label/configuration), then one would also need to manually incorporate all those global changes in the resulting rewriting logic definition.

We first show that any MSOS proof system, say MSOS, can be mechanically translated into a rewriting logic theory, say $\mathcal{R}_{\text{MSOS}}$, in such a way that two important aspects of the original MSOS definition are preserved: 1) the corresponding derivation relations are step-for-step equivalent, that is, $\text{MSOS} \vdash C \to C'$ if and only if $\mathcal{R}_{\text{MSOS}} \vdash \mathcal{R}_{C \to C'}$, where $\mathcal{R}_{C \to C'}$ is the corresponding syntactic translation of the MSOS sequent $C \to C'$ into a rewriting logic sequent; and 2) $\mathcal{R}_{\text{MSOS}}$ is as modular as MSOS. Second, we apply our generic translation technique on the MSOS formal system MSOS(IMP) defined in Section 3.4.1 and obtain a rewriting logic semantics of IMP that is step-for-step equivalent to and as modular as MSOS(IMP). The modularity of MSOS(IMP) and of $\mathcal{R}_{\text{MSOS(IMP)}}$ will pay off when we extend IMP in Section 3.8. Finally, we show how $\mathcal{R}_{\text{MSOS(IMP)}}$ can be seamlessly defined in Maude, yielding another interpreter for IMP (in addition to those corresponding to the big-step and small-step SOS definitions of IMP in Sections 3.2.3 and 3.3.3).

Computationally and Modularly Faithful Embedding of MSOS into Rewriting Logic

Our embedding of MSOS into rewriting logic is very similar to that of small-step SOS into rewriting logic, with one important exception: the non-syntactic components of the configuration are all grouped into a *record*, which is a multiset of *attributes*, each attribute being a pair associating appropriate semantic information to a *field*. This allows us to use multiset *matching* (or matching modulo associativity, commutativity, and identity) in the corresponding rewrite rules to extract the needed semantic information from the record, thus achieving not only a computationally equivalent embedding of MSOS into rewriting logic, but also one with the same degree of modularity as MSOS.

Formally, let us assume an arbitrary MSOS formal proof system. Let *Attribute* be a fresh sort and let *Record* be the sort **Bag{***Attribute***}** (that means that we assume all the infrastructure needed to define records as comma-separated bags, or multisets, of attributes). For each field *Field* holding semantic contents *Contents* that appears unprimed or primed in any of the labels on any of the transitions in any of the rules of the MSOS proof system, let us assume an operation

"*Field* = _ : *Contents* → *Attribute*" (the name of this postfix unary operation is "*Field* = _"). Finally, for each syntactic category *Syntax* used in any transition that appears anywhere in the MSOS proof system, let us define a configuration construct "$\langle \_, \_ \rangle$: *Syntax* × *Record* → *Configuration*". We are now ready to define our transformation of an MSOS rule into a rewriting logic rule:

1. Translate it into an SOS rule, as discussed right above Section 3.4.1; we could also go directly from MSOS rules to rewriting logic rules, but we would have to repeat most of the steps from MSOS to SOS that were already discussed;

2. Group all the semantic components in the resulting SOS configurations into a corresponding record, where each semantic component translates into a corresponding attribute using a corresponding label;

3. If the same attributes appear multiple times in a rule and their semantic components are not used anywhere in the rule, then replace them by a generic variable of sort *Record*;

4. Finally, use the technique in Section 3.3.3 to transform the resulting SOS-like rules into rewrite rules, tagging the left-hand-side configurations with the ∘ symbol.

Applying the steps above, the four MSOS rules discussed right above Section 3.4.1 (namely the ones for variable assignment, output, sequential composition of statements, and variable lookup) translate into the following rewriting logic rules:

$$\begin{aligned}
\circ\langle X := I, (\mathsf{state} = \sigma, \rho)\rangle &\to \langle\, \mathtt{skip}\,, (\mathsf{state} = \sigma[I/X], \rho)\rangle \\
\circ\langle\, \mathtt{output}\,(i), (\mathsf{output} = \omega,\ \rho)\rangle &\to \langle\, \mathtt{skip}\,, (\mathsf{output} = \omega i,\ \rho)\rangle \\
\circ\langle S_1\ ;\ S_2, \rho\rangle &\to \langle S_1'\ ;\ S_2, \rho'\rangle \quad \textbf{if} \quad \circ\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle \\
\circ\langle X, (\mathsf{state} = \sigma, \rho)\rangle &\to \langle \sigma(X), (\mathsf{state} = \sigma, \rho)\rangle
\end{aligned}$$

We use the same mechanism as for small-step SOS to obtain the reflexive and transitive many-step closure of the MSOS one-step transition relation. This mechanism was discussed in detail in Section 3.3.3; it essentially consists of adding a configuration marker $\star$ together with a rule "$\star C \to \star C'$ **if** $\circ\, C \to C'$" iteratively applying the one-step relation.

Theorem 5. ***(Faithful embedding of MSOS into rewriting logic)*** *For any MSOS definition* MSOS*, and any appropriate configurations C and C', the following equivalences hold:*

$$\text{MSOS} \vdash C \to C' \iff \mathcal{R}_{\text{MSOS}} \vdash \circ\, \overline{C} \to^1 \overline{C'} \iff \mathcal{R}_{\text{MSOS}} \vdash \circ\, \overline{C} \to \overline{C'}$$

$$\text{MSOS} \vdash C \to^\star C' \iff \mathcal{R}_{\text{MSOS}} \vdash \star\, \overline{C} \to \star\, \overline{C'}$$

where $\mathcal{R}_{\text{MSOS}}$ is the rewriting logic semantic definition obtained from MSOS *by translating each rule in* MSOS *as above. (Recall from Section 2.7 that $\to^1$ is the one-step rewriting relation obtained by dropping the reflexivity and transitivity rules of rewriting logic. Also, as C and C' are parameter-free —parameters only appear in rules— $\overline{C}$ and $\overline{C'}$ are ground terms.)*

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, let us elaborate on the apparent differences between MSOS and $\mathcal{R}_{\text{MSOS}}$ from a user perspective. The most visible difference is the SOS-like style of writing the rules, namely using configurations instead of labels, which also led to the inheritance of the ∘ mechanism from the embedding of SOS into rewriting logic. Therefore, the equivalent rewriting logic definition is slightly more verbose than the

sorts:
$\mathit{Attribute}$, Record, $\mathit{Configuration}$, $\mathit{ExtendedConfiguration}$
subsorts and aliases:
$\mathit{Record} = \mathbf{Bag}\{\mathit{Attribute}\}$
$\mathit{Configuration} < \mathit{ExtendedConfiguration}$
operations:

$$\begin{array}{rcll} \mathsf{state} = \_ & : & \mathit{State} \to \mathit{Attribute} & // \text{ more "fields" can be added by need} \\ \langle \_, \_ \rangle & : & \mathit{AExp} \times \mathit{Record} \to \mathit{Configuration} & \\ \langle \_, \_ \rangle & : & \mathit{BExp} \times \mathit{Record} \to \mathit{Configuration} & \\ \langle \_, \_ \rangle & : & \mathit{Stmt} \times \mathit{Record} \to \mathit{Configuration} & \\ \langle \_ \rangle & : & \mathit{Pgm} \to \mathit{Configuration} & \\ \circ \_ & : & \mathit{Configuration} \to \mathit{ExtendedConfiguration} & // \text{ reduce one step} \\ \star \_ & : & \mathit{Configuration} \to \mathit{ExtendedConfiguration} & // \text{ reduce all steps} \end{array}$$

rule:
$\star\, C \to \star\, C'$ **if** $\circ\, C \to C'$ // where C, C' are variables of sort $\mathit{Configuration}$

Figure 3.22: Configurations and infrastructure for the rewriting logic embedding of MSOS(IMP).

original MSOS definition. On the other hand, it has the advantage that it is more direct than the MSOS definition, in that it eliminates all the notational conventions. Indeed, if we strip MSOS out of its notational conventions and go straight to its essence, we find that that essence is precisely its use of multiset matching to modularly access the semantic components of the configuration. MSOS chose to do this on the labels, using specialized conventions for read-only/write-only/read-write components, while our rewriting logic embedding of MSOS does it in the configurations, uniformly for all semantic components. Where precisely this matching takes place is, in our view, less relevant. What is relevant and brings MSOS its modularity is that multiset matching *does* happen. Therefore, similarly to the big-step and small-step SOS representations in rewriting logic, we conclude that the rewrite theory $\mathcal{R}_{\mathrm{MSOS}}$ *is* MSOS, and *not* an "encoding" of it.

Like for the previous embeddings of big-step and small-step SOS into rewriting logic, unfortunately, $\mathcal{R}_{\mathrm{MSOS}}$ (and implicitly MSOS) still lacks the main strengths of rewriting logic, namely context-insensitive and parallel application of rewrite rules. Indeed, the rules of $\mathcal{R}_{\mathrm{MSOS}}$ can only apply at the top, sequentially, so these rewrite theories corresponding to the faithful embedding of MSOS follow a rather poor rewriting logic style. Like for the previous embeddings, this is not surprising though and does not question that quality of our embeddings. All it says is that MSOS was not meant to have the capabilities of rewriting logic w.r.t. context-insensitivity and parallelism; indeed, all MSOS tries to achieve is to address the lack of modularity of SOS and we believe that it succeeded in doing so. Unfortunately, SOS has several other major problems.

MSOS of IMP in Rewriting Logic

We here discuss the complete MSOS definition of IMP in rewriting logic, obtained by applying the faithful embedding technique discussed above to the MSOS definition of IMP in Section 3.4.1. Figure 3.22 gives an algebraic definition of configurations as well as needed additional record infrastructure; all the sorts, operations, and rules in Figure 3.22 were already discussed either above or in Section 3.3.3. Figure 3.23 gives the rules of the rewriting logic theory $\mathcal{R}_{\mathrm{MSOS(IMP)}}$ that is

$$\circ\langle X, (\mathsf{state} = \sigma, \rho)\rangle \to \langle \sigma(X), (\mathsf{state} = \sigma, \rho)\rangle$$

$$\begin{array}{rll}
\circ\langle A_1 + A_2, \rho\rangle \to \langle A_1' + A_2, \rho'\rangle & \textbf{if} & \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle A_1 + A_2, \rho\rangle \to \langle A_1 + A_2', \rho'\rangle & \textbf{if} & \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 + I_2, \rho\rangle \to \langle I_1 +_{Int} I_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle A_1 \,/\, A_2, \rho\rangle \to \langle A_1' \,/\, A_2, \rho'\rangle & \textbf{if} & \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle A_1 \,/\, A_2, \rho\rangle \to \langle A_1 \,/\, A_2', \rho'\rangle & \textbf{if} & \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 \,/\, I_2, \rho\rangle \to \langle I_1 \,/_{Int}\, I_2, \rho\rangle & \textbf{if} & I_2 \neq 0
\end{array}$$

$$\begin{array}{rll}
\circ\langle A_1 \mathsf{<=} A_2, \rho\rangle \to \langle A_1' \mathsf{<=} A_2, \rho'\rangle & \textbf{if} & \circ\langle A_1, \rho\rangle \to \langle A_1', \rho'\rangle \\
\circ\langle I_1 \mathsf{<=} A_2, \rho\rangle \to \langle I_1 \mathsf{<=} A_2', \rho'\rangle & \textbf{if} & \circ\langle A_2, \rho\rangle \to \langle A_2', \rho'\rangle \\
\circ\langle I_1 \mathsf{<=} I_2, \rho\rangle \to \langle I_1 \;\leq_{Int}\; I_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle\, \mathtt{not}\;\; B, \rho\rangle \to \langle\, \mathtt{not}\;\; B', \rho'\rangle & \textbf{if} & \circ\langle B, \rho\rangle \to \langle B', \rho'\rangle \\
\circ\langle\, \mathtt{not\ true}\,, \rho\rangle \to \langle\, \mathtt{false}\,, \rho\rangle & & \\
\circ\langle\, \mathtt{not\ false}\,, \rho\rangle \to \langle\, \mathtt{true}\,, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle B_1 \,\mathtt{and}\, B_2, \rho\rangle \to \langle B_1' \,\mathtt{and}\, B_2, \rho'\rangle & \textbf{if} & \circ\langle B_1, \rho\rangle \to \langle B_1', \rho'\rangle \\
\circ\langle\, \mathtt{false}\,\mathtt{and}\, B_2, \rho\rangle \to \langle\, \mathtt{false}\,, \rho\rangle & & \\
\circ\langle\, \mathtt{true}\,\mathtt{and}\, B_2, \rho\rangle \to \langle B_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle X := A, \rho\rangle \to \langle X := A', \rho'\rangle & \textbf{if} & \circ\langle A, \rho\rangle \to \langle A', \rho'\rangle \\
\circ\langle X := I, (\mathsf{state} = \sigma, \rho)\rangle \to \langle\, \mathtt{skip}\,, (\mathsf{state} = \sigma[I/X], \rho)\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle S_1 \;;\; S_2, \rho\rangle \to \langle S_1' \;;\; S_2, \rho'\rangle & \textbf{if} & \circ\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle \\
\circ\langle\, \mathtt{skip}\;;\; S_2, \rho\rangle \to \langle S_2, \rho\rangle & &
\end{array}$$

$$\begin{array}{rll}
\circ\langle\, \mathtt{if}\, B\, \mathtt{then}\, S_1\, \mathtt{else}\, S_2, \rho\rangle \to \langle\, \mathtt{if}\, B'\, \mathtt{then}\, S_1\, \mathtt{else}\, S_2, \rho'\rangle & \textbf{if} & \circ\langle B, \rho\rangle \to \langle B', \rho'\rangle \\
\circ\langle\, \mathtt{if\ true\ then}\, S_1\, \mathtt{else}\, S_2, \rho\rangle \to \langle S_1, \rho\rangle & & \\
\circ\langle\, \mathtt{if\ false\ then}\, S_1\, \mathtt{else}\, S_2, \rho\rangle \to \langle S_2, \rho\rangle & &
\end{array}$$

$$\circ\langle\, \mathtt{while}\, B\, \mathtt{do}\, S, \rho\rangle \to \langle\, \mathtt{if}\, B\, \mathtt{then}\, (S \;;\; \mathtt{while}\, B\, \mathtt{do}\, S)\, \mathtt{else\ skip}\,, \rho\rangle$$

$$\circ\langle\, \mathtt{vars}\, Xl \;;\; S\rangle \to \langle S, (\mathsf{state} = Xl \mapsto 0)\rangle$$

Figure 3.23: $\mathcal{R}_{\mathrm{MSOS(IMP)}}$: the complete MSOS of IMP in rewriting logic.

obtained by applying the procedure above to the MSOS of IMP in Figures 3.20 and 3.21. Like before, we used the rewriting logic convention that variables start with upper-case letters; if they are greek letters, then we use a similar but larger symbol (e.g., Σ instead of σ for variables of sort *State*, or $\Rho$ instead of ρ for variables of sort *Record*). The following corollary of Theorem 5 establishes the faithfulness of the representation of the MSOS of IMP in rewriting logic:

Corollary 4. MSOS(IMP) $\vdash C \to C' \iff \mathcal{R}_{\text{MSOS(IMP)}} \vdash \circ\overline{C} \to \overline{C'}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system MSOS(IMP) and generic rewriting logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{MSOS(IMP)}}$; the two are faithfully equivalent. Moreover, by the discussion following Theorem 5, $\mathcal{R}_{\text{MSOS(IMP)}}$ is also as modular as MSOS(IMP). This will be further emphasized in Section 3.8, where we will extend IMP with several features, some of which requiring more attributes.

☆ Maude Definition of IMP MSOS

Figure 3.24 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{MSOS(IMP)}}$ in Figures 3.23 and 3.22. The Maude module `IMP-SEMANTICS-MSOS` in Figure 3.24 is executable, so Maude, through its rewriting capabilities, yields an MSOS interpreter for IMP the same way it yielded big-step and small-step SOS interpreters in Sections 3.2.3 and 3.3.3, respectively; for example, the Maude rewrite command

```
rewrite * < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

```
rewrites: 6335 in ... cpu (... real) (... rewrites/second)
result ExtendedConfiguration: * < skip, state = (n |-> 0 , s |-> 5050) >
```

Note that the rewrite command above took the same number of rewrite steps as the similar command executed on the small-step SOS of IMP in Maude discussed in Section 3.3.3, namely 6335. This is not unexpected, because matching is not counted as rewrite steps, no matter how complex it is.

Like for the big-step and small-step SOS definitions in Maude, one can also use any of the general-purpose tools provided by Maude on the MSOS definition above. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
search * < sumPgm > =>! Cfg:ExtendedConfiguration .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic. The same number of states as in the case of small-step SOS will be generated by this search command, namely 1509.

3.4.3 Notes

Modular Structural Operational Semantics (MSOS) was introduced in 1999 by Peter Mosses [55] and since then mainly developed by himself and his collaborators (e.g., [56, 57, 58]). In this section we used the implicitly-modular variant of MSOS introduced in [58], which, as acknowledged by the authors of [58], was partly inspired from discussions with us[2]. To be more precise, we used a slightly

[2]In fact, drafts of this book that preceded [58] dropped the implicit labels in MSOS rules for notational simplicity; Peter Mosses found that simple idea worthwhile formalizing within MSOS.

```
mod IMP-CONFIGURATIONS-MSOS is including IMP-SYNTAX + STATE .
  sorts Attribute Record Configuration ExtendedConfiguration .
  subsort Attribute < Record .
  subsort Configuration < ExtendedConfiguration .
  op empty : -> Record .
  op _,_ : Record Record -> Record [assoc comm id: empty] .
  op state`=_ : State -> Attribute .
  op <_,_> : AExp Record -> Configuration .
  op <_,_> : BExp Record -> Configuration .
  op <_,_> : Stmt Record -> Configuration .
  op <_> : Pgm -> Configuration .
  op o_ : Configuration -> ExtendedConfiguration [prec 80] .   --- one step
  op *_ : Configuration -> ExtendedConfiguration [prec 80] .   --- all steps
  var Cfg Cfg' : Configuration .
 crl * Cfg => * Cfg' if o Cfg => Cfg' .
endm

mod IMP-SEMANTICS-MSOS is including IMP-CONFIGURATIONS-MSOS .
  var X : Id .   var R R' : Record . var Sigma Sigma' : State .
  var I I1 I2 : Int .  var Xl : List{Id} .  var S S1 S1' S2 : Stmt .
  var A A' A1 A1' A2 A2' : AExp .  var B B' B1 B1' B2 B2' : BExp .

  rl o < X,(state = Sigma, R) > => < Sigma(X),(state = Sigma, R) > .
 crl o < A1 + A2,R > => < A1' + A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 + A2,R > => < A1 + A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 + I2,R > => < I1 +Int I2,R > .
 crl o < A1 / A2,R > => < A1' / A2,R' > if o < A1,R > => < A1',R' > .
 crl o < A1 / A2,R > => < A1 / A2',R' > if o < A2,R > => < A2',R' > .
 crl o < I1 / I2,R > => < I1 /Int I2,R > if I2 =/= 0 .
 crl o < A1 <= A2,R > => < A1' <= A2,R' > if o < A1,R > => < A1',R' > .
 crl o < I1 <= A2,R > => < I1 <= A2',R' > if o < A2,R > => < A2',R' > .
  rl o < I1 <= I2,R > => < I1 <=Int I2,R > .
 crl o < not B,R > => < not B',R' > if o < B,R > => < B',R' > .
  rl o < not  true,R > => < false,R > .
  rl o < not false,R > => <  true,R > .
 crl o < B1 and B2,R > => < B1' and B2,R' > if o < B1,R > => < B1',R' > .
  rl o < false and B2,R > => < false,R > .
  rl o <  true and B2,R > => < B2,R > .
 crl o < X := A,R > => < X := A',R' > if o < A,R > => < A',R' > .
  rl o < X := I,(state = Sigma, R) > => < skip,(state = Sigma[I / X], R) > .
 crl o < S1 ; S2,R > => < S1' ; S2,R' > if o < S1,R > => < S1',R' > .
  rl o < skip ; S2,R > => < S2,R > .
 crl o < if B then S1 else S2,R > => < if B' then S1 else S2,R' > if o < B,R > => < B',R'  > .
  rl o < if true then S1 else S2,R > => < S1,R > .
  rl o < if false then S1 else S2,R > => < S2,R > .
  rl o < while B do S,R > => < if B then (S ; while B do S) else skip,R > .
  rl o < vars Xl ; S > => < S,(state = Xl |-> 0) > .
endm
```

Figure 3.24: The MSOS of IMP in Maude, including the definition of configurations.

simplified version of implicitly-modular MSOS here. In MSOS in its full generality, one can also declare some transitions "unobservable"; to keep the presentation simpler, we here omitted all the observability aspects of MSOS.

The idea of our representation of MSOS into rewriting logic adopted in this section is taken over from [72]. At our knowledge, [45] gives the first representation of MSOS into rewriting logic. The representation in [45] also led to the development of the Maude MSOS tool [18]. What is different in the representation in [45] from ours is that the former uses two different types of configuration wrappers, one for the left-hand side of the transitions and one for the right-hand side; this was already discussed in Section 3.3.4.